The Paxos Algorithm for Fault Tolerance in Distributed Systems

By Noah Redden

COMP 3010
Robert Guderian
University of Manitoba
Winnipeg, Manitoba
2021-04-14

Distributed systems are widely used in our daily life; They use the combined throughput of multiple computers to drive services for finance, shopping, infrastructure, and entertainment. Many companies employ a distributed system, including Google, Microsoft, Amazon, and Netflix. These companies need their systems to be properly functioning around the clock, otherwise, a loss of profit will occur and their email boxes may be filled with the rantings of irate customers. One of the main problems in maintaining a network of distributed systems is ensuring that the individual machines keep reliable communication between themselves, even if one or more of the machines suffer from a complete failure. But how can we ensure that these individual failures are tolerated by the system to prevent them from bringing down the rest of the machines? The answer is the Paxos algorithm. The Paxos algorithm is exceptional for fault tolerance in a distributed system. It can be used in a distributed system to ensure that communication between machines is reliable and fault-tolerant. Using Paxos can allow Google, Microsoft, Amazon, Netflix, etc. to continue to make their obscenely large profit margins and keep their customers happy. In the following paper, we will explore the definition of fault tolerance and why we need it for distributed systems, an explanation of the Paxos algorithm, all its aspects and how it works, and the main reasons why the Paxos algorithm is exceptional for fault tolerance in our distributed systems.

What is fault tolerance, and why is it important? Zhao (2014) describes fault tolerance as an approach to improving the dependability of distributed systems by ensuring that the system is able to recover from faults without an interruption in service. While ideally, we could just make systems that never fail, it could be considered naive to think that any system is completely fault-proof, and it is always a good idea to plan for something going extremely wrong in a system. A fault-tolerant system should be able to have some components fail while minimizing the impacts of faults on the system as a whole (Zhao, 2014). When things break, the average users of the system should not be able to tell that anything is amiss, and hopefully, the fault tolerance will allow the users to continue using the system to browse, shop, watch movies, etc.

One way we can keep the entire system from crashing and maintaining the user’s illusion of a perfectly functioning system is by using multiple redundant machines.

So long as redundant machines fail independently of each other, failures can be detected and a faulty machine can be swapped out for a non-faulty redundant machine that picks up shortly before where the faulty machine left off (Zhao, 2014). The redundant machine may use a set of alternate programs that aim to accomplish the same output but may be less likely to produce a fault (Zhao, 2014), further increasing a system's fault tolerance. Thus, it seems that using multiple redundant machines is a good method for increasing fault tolerance. However, this method can have its drawbacks as well.

A fault that could occur to these multiple machines can involve multiple clients sending commands to the system which arrive at different individual machines in different orders (Van Renesse and Altinbuken, 2015). This may result in the machines producing different outputs from each other, and the machines may end up in different states from each other when they should all be on the same page (Van Renesse and Altinbuken, 2015).

For example, if User A visits their favourite online store. They find a shiny new graphics card, with only 1 left in stock. Wanting to upgrade their old computer, they click the buy button. However, User B also saw the graphics card at the same time as User B, and they also clicked the buy button, though ever so slightly after User A did. It is theoretically possible that User B's purchase will come in first on one machine while on another machine it registers that User A came in first, creating a predicament where the whole system is unsure which user would be able to purchase the graphics card. Thus, we must ensure that the entire system is in agreement as to what commands come in and in what order.

How can we ensure that conflicting machines will not be a problem for our distributed system? It is still valuable to keep multiple machines for the overall benefits in fault tolerance and redundancy they bring, so scrapping them would not be an option. But to keep our multiple machines, we need something in place that prevents the individual machines from giving different outputs and states from one another.

Enter the Paxos algorithm, our answer to keeping machines in agreement. The name Paxos comes from the Greek island of Paxos, which was known for having an early democratic parliament (Lamport, 1998). The voting citizens of Paxos were less

interested in participating in parliament and more interested in trade and commerce, resulting in a part-time parliament where legislators were frequently absent from proceedings (Lamport, 1998). To ensure that absent legislators knew about what laws were passed, a system of messengers was devised to inform everyone (Lamport, 1998). The absences of these legislators form a parallel to distributed machines suffering from failures, and the Paxos parliament's solution can be applied similarly (Lamport, 1998). Others in the world of distributed computing have seen the uses of such a system, such as Google's Chubby system, which is an implementation of Paxos used for various programs, including Google File Systems and Bigtable clients (Zhao, 2014).

Paxos is a consensus algorithm, which means that the collection of machines that are using this algorithm must all agree or come to a consensus about whether or not a proposed value is chosen (Lamport, 2001). This proposed value, if accepted by the machines, will affect the state and output of the machines.

The value can be proposed by specific machines in the system, which has a designation of the proposer (Lamport, 2001). There are several designations for machines, the proposer being one. Proposers, as the name implies, propose a value to the other machines (Lamport, 2001). This value could be from a user who wants to do something with the system. Proposers can also make multiple proposals and cancel a proposal before it is complete (Lamport, 2001).

That proposed value is sent to a machine that is designated as an acceptor. It is the job of the acceptor machine to accept or not accept the proposed value (Lamport, 2001). Multiple acceptors are used, since having only a single acceptor would have no fault tolerance for that single acceptor failing (Zhao, 2014). For a proposed value to be chosen, a large enough number of acceptors must accept it, usually a majority of machines in a system (Lamport, 2001). Proposed values are assigned a number, known as a proposal number, that is used by acceptors to keep track of all the different proposals they may encounter (Lamport, 2001). When a value is chosen from a proposal of proposal number n, all proposals with proposal numbers of a higher amount have the same value (Lamport, 2001). This is done to maintain that only one value is chosen (Lamport, 2001).

When a proposal is issued, a few things will happen. A proposer will get a new proposal number *n*, then requests from acceptors to never accept proposals of a lower proposal value, and also requests the highest-numbered proposal that is less than *n* (Lamport, 2001). If the proposer receives enough responses from acceptors, a proposal is issued with proposal number *n* and the value from either the highest valued proposal number from the responses or the proposer themselves (Lamport, 2001). The proposal is then sent out to acceptors, requesting that they accept the proposal (Lamport, 2001). When an acceptor receives this request, it can accept the request, so long as it has not received any request of a proposal number greater than *n* (Lamport, 2001). Once a majority of acceptors have accepted the value, then that value is now chosen and the proposal is successful (Lamport, 2001). If the proposal is unsuccessful, then the proposer is notified and no changes are made (Lamport, 2001). Regardless of the outcome, a proposer is more than free to make more proposals if they so desire, and the algorithm starts over again (Lamport, 2001).

There is a final designation in the Paxos algorithm, it is that of a learner. Learners are machines that will later learn about a chosen value after it has been accepted by the prerequisite majority of acceptors (Lamport, 2001). Learners can be machines that failed or went down for maintenance that now need to learn what has happened in the system since they left. There are a few ways to do this; One way would be for an acceptor to notify all learners that a majority have accepted a value, but this could be expensive for the acceptor to do (Lamport, 2001). Having all the learners periodically poll the acceptors to see if they have accepted a new value is also possible, though having every learner conduct polling could be taxing due to the number of messages it would create in the system (Zhao, 2014). It could be more efficient to have one learner act as a “designated learner”, who is the only learner informed by acceptors about which value they had accepted (Lamport, 2001). The designated learner would then inform the other learners about this value that has been chosen, which would require fewer messages between acceptors and learners (Lamport, 2001).

There are problems with this approach, as it requires extra time for the designated learner to send messages to the other learners, and more importantly, this approach can suffer from reliability if the designated learner fails for whatever reason

(Lamport, 2001). This second problem can be remedied by having multiple designated learners, each of which can work to tell all learners about the accepted value that the acceptors decided on (Lamport, 2001). This remedy does restore reliability, but at the cost of complexity in communication and implementation (Lamport, 2001).

It may be beneficial for the system that a learner ensures that the value they have learned is the correct one that has been accepted. Doing so would help defend the system against any incorrect values, innocent or otherwise. Confirming a value can be simple for a learner; It can ask a proposer to re-propose that value (Zhao, 2014). The result of the proposal would confirm if that value was chosen or not (Zhao, 2014).

For efficiency's sake, there should also be a designated proposer as well. This machine will usually be the first machine to try and communicate with the acceptors and send proposals (Lamport, 2001). This prevents multiple proposers from trying to send out competing requests that override each other and preventing values from being chosen (Lamport, 2001).

With all these roles, the Paxos algorithm can properly function and work to achieve consensus. The designated proposer will send out requests to the acceptors. If enough acceptors respond, this is followed by a proposal that can be accepted or not. Once the value is accepted by a majority, the designated learners will be sent the value and knowledge of its acceptance. The designated learners will send out this information to the rest of the learners. And with that, the values are accepted and everybody knows them. But this all seems like a lot of work for accepting a single value. Is the Paxos algorithm's fault tolerance effective enough to make all this effort, time, and resource use worth it?

Paxos' fault tolerance is very effective. It is very important for our distributed system that should any problem occur that it can be dealt with in a way such that the damages are minimized. If a machine goes down, it should not bring the rest of the system with it. The way Paxos handles these faults makes it worth the effort and complexity of implementation.

Paxos is effective for having downed machines catch up to the others. When a machine is brought down due to maintenance or faults, it needs to be taught what happened in its absence to properly rejoin the system. Having this contributes to fault

tolerance, as it allows machines to be brought back up after failures and become redundant for the machines that operated normally in its absence. Since machines in a Paxos system can be designated as a learner (Lamport, 2001), we can designate machines that were just brought back up or possibly back up for the first time. Acceptors can then notify the learners (either by notifying every learner machine or just notifying the designated learners that then, in turn, notify the others) what values were accepted by the system before the learners were brought online (Lamport, 2001). Having this option for machines to learn and catch up to the other machines in the rest of the system is vital to a well-functioning system.

Of course, one of the most obvious aspects of Paxos' fault tolerance is its tolerance for machines being disabled. This is the main draw of using multiple redundant machines after all. Paxos protocol can easily tolerate machines failing, including scenarios where a minority of acceptors fail (Van Renesse and Altinbuken, 2015). If somebody accidentally trips on a wire and unplugs a machine, users are unlikely to notice any drop in the quality of the system due to the many redundant machines. As a reminder, these machines can be brought back up to speed on what values have been accepted during their absence (Lamport, 2001), so long-term effects of a system being disabled or non-existent. This toleration of downed systems is one of the strongest points of a distributed system, and systems with Paxos are no exception to this.

Paxos can also be used to detect arbitrary faults. During the operations of the Paxos algorithm, the outputs of involved machines can be monitored for deviations from expected values (Barbieri, dos Santos, and Maciel Dias Vieira, 2020). When designing fault tolerance for systems it usually is rather helpful to identify when a fault occurs. If we know that a fault has occurred on one of the machines we can act to stop that fault from affecting the entire system and bringing it down. Faults in Paxos can be hard to detect, especially since data can be corrupted in a way that does not seem out of place to the algorithm since the messages used to send this corrupt data were sent correctly as far as Paxos is concerned (Barbieri, dos Santos, and Maciel Dias Vieira, 2020). Implementation of checks for data integrity, state corruptions, and programmer semantics can be used to great effect to detect faults and invariants in Paxos (Barbieri,

dos Santos, and Maciel Dias Vieira, 2020). Doing so can be useful in protecting precious data from arbitrary faults.

Paxos also solves another problem. One of the main problems we had with our distributed system was that the multiple machines could have multiple users send them conflicting commands concurrently, resulting in different outputs from the different machines and general confusion. We know from earlier that Paxos only lets one proposal get through at a time (Lamport, 2001). Paxos ensures that all commands are processed in the same order and that the distributed system acts similarly in output to a single machine (Van Renesse and Altinbuken, 2015). Thus, in that example from earlier, where customer A is trying to buy the last graphics card before customer B can, the system will go through the process of accepted customer A's request before customer B can make a request, which prevents any system failures as a result of two concurrent purchases of the same product. Reducing the situations in which our system crashes or where the output varies wildly from machine to machine is always helpful, and the fault tolerance afforded by Paxos in this situation does exactly that.

In conclusion, Paxos is efficient for fault tolerance in a distributed system. Paxos allows downed machines to catch up to the rest of the machines and learn what has happened with the state of the system in their absence. Paxos can tolerate machines failing completely and catastrophically without any overall harm to the system, and certainly not any harm to the system that can be detected by a user. Paxos can be configured to combine with arbitrary fault detection to protect data from data corruption. And, most importantly, Paxos solves the problem of multiple users connecting concurrently to a distributed system and risking different outputs from different machines by only letting one proposal in at a time. All these features are worthwhile to implement, as they result in a strong system that is capable of surviving common faults that plague ordinary distributed systems and bring them down. This means that Google, Microsoft, Amazon, Netflix, and many more companies can provide exemplary services in advertising, software, shopping, and media consumption, respectively, to their customers without fear of machine crashes interrupting their services and costing their profits millions or possibly billions in revenue. It also means that the users of these services never even have to know or think about how they are using multiple machines

at once or worry about if a single machine or a few machines crash unexpectedly during daily use. The Paxos algorithm provides exceptional fault tolerance that allows our distributed system to run these tasks every day with minimized risk of the entire system succumbing to faults and crashes.

References

Barbieri, R., dos Santos, E. and Maciel Dias Vieira, G., 2020. ‘Decentralized Validation for Non-malicious Arbitrary Fault Tolerance in Paxos’.

Lamport, L., 1998. *The part-time parliament*.

Lamport, L., 2001. *Paxos Made Simple*.

Van Renesse, R. and Altinbuken, D., 2015. ‘Paxos Made Moderately Complex’, *ACM Computing Surveys*, 47(3), pp.1-36.

Zhao, W., 2014. *Building Dependable Distributed Systems*. Scrivener Publishing, pp.9-13.